modern compiler implementation in Java

second edition

andrew w. appel

CAMBRIDGE

more information - www.cambridge.org/052182060X

This page intentionally left blank

Modern Compiler Implementation in Java
Second Edition

This textbook describes all phases of a compiler: lexical analysis, parsing, abstract syntax, semantic actions, intermediate representations, instruction selection via tree matching, dataflow analysis, graph-coloring register allocation, and runtime systems. It includes good coverage of current techniques in code generation and register allocation, as well as the compilation of functional and object-oriented languages, which is missing from most books. The most accepted and successful techniques are described concisely, rather than as an exhaustive catalog of every possible variant. Detailed descriptions of the interfaces between modules of a compiler are illustrated with actual Java classes.

The first part of the book, Fundamentals of Compilation, is suitable for a one-semester first course in compiler design. The second part, Advanced Topics, which includes the compilation of object-oriented and functional languages, garbage collection, loop optimization, SSA form, instruction scheduling, and optimization for cache-memory hierarchies, can be used for a second-semester or graduate course.

This new edition has been rewritten extensively to include more discussion of Java and object-oriented programming concepts, such as visitor patterns. A unique feature is the newly redesigned compiler project in Java for a subset of Java itself. The project includes both front-end and back-end phases, so that students can build a complete working compiler in one semester.

Andrew W. Appel is Professor of Computer Science at Princeton University. He has done research and published papers on compilers, functional programming languages, runtime systems and garbage collection, type systems, and computer security; he is also author of the book *Compiling with Continuations.* He is a designer and founder of the Standard ML of New Jersey project. In 1998, Appel was elected a Fellow of the Association for Computing Machinery for "significant research contributions in the area of programming languages and compilers" and for his work as editor-in-chief (1993–97) of the *ACM Transactions on Programming Languages and Systems,* the leading journal in the field of compilers and programming languages.

Jens Palsberg is Associate Professor of Computer Science at Purdue University. His research interests are programming languages, compilers, software engineering, and information security. He has authored more than 50 technical papers in these areas and a book with Michael Schwartzbach, *Object-oriented Type Systems.* In 1998, he received the National Science Foundation Faculty Early Career Development Award, and in 1999, the Purdue University Faculty Scholar award.

Modern Compiler Implementation in Java

Second Edition

ANDREW W. APPEL
Princeton University

with JENS PALSBERG
Purdue University

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published in printed format 2002

ISBN 0-511-03930-1 eBook (Adobe Reader)
ISBN 0-521-82060-X hardback

Contents

Preface ix

Part I Fundamentals of Compilation

1 Introduction 3
1.1 Modules and interfaces 4
1.2 Tools and software 5
1.3 Data structures for tree languages 7

2 Lexical Analysis 16
2.1 Lexical tokens 17
2.2 Regular expressions 18
2.3 Finite automata 21
2.4 Nondeterministic finite automata 24
2.5 Lexical-analyzer generators 30

3 Parsing 38
3.1 Context-free grammars 40
3.2 Predictive parsing 45
3.3 LR parsing 55
3.4 Using parser generators 68
3.5 Error recovery 76

4 Abstract Syntax 86
4.1 Semantic actions 86
4.2 Abstract parse trees 89
4.3 Visitors 93

5 Semantic Analysis 103
5.1 Symbol tables 103

5.2 Type-checking MiniJava 111

6 Activation Records 116
6.1 Stack frames 118
6.2 Frames in the MiniJava compiler 126

7 Translation to Intermediate Code 136
7.1 Intermediate representation trees 137
7.2 Translation into trees 140
7.3 Declarations 155

8 Basic Blocks and Traces 162
8.1 Canonical trees 163
8.2 Taming conditional branches 169

9 Instruction Selection 176
9.1 Algorithms for instruction selection 179
9.2 CISC machines 187
9.3 Instruction selection for the MiniJava compiler 190

10 Liveness Analysis 203
10.1 Solution of dataflow equations 205
10.2 Liveness in the MiniJava compiler 214

11 Register Allocation 219
11.1 Coloring by simplification 220
11.2 Coalescing 223
11.3 Precolored nodes 227
11.4 Graph-coloring implementation 232
11.5 Register allocation for trees 241

12 Putting It All Together 249

Part II Advanced Topics

13 Garbage Collection 257
13.1 Mark-and-sweep collection 257
13.2 Reference counts 262
13.3 Copying collection 264

13.4 Generational collection 269
13.5 Incremental collection 272
13.6 Baker's algorithm 274
13.7 Interface to the compiler 275

14 Object-Oriented Languages 283
14.1 Class extension 283
14.2 Single inheritance of data fields 284
14.3 Multiple inheritance 286
14.4 Testing class membership 289
14.5 Private fields and methods 292
14.6 Classless languages 293
14.7 Optimizing object-oriented programs 293

15 Functional Programming Languages 298
15.1 A simple functional language 299
15.2 Closures 301
15.3 Immutable variables 302
15.4 Inline expansion 308
15.5 Closure conversion 316
15.6 Efficient tail recursion 319
15.7 Lazy evaluation 321

16 Polymorphic Types 335
16.1 Parametric polymorphism 336
16.2 Polymorphic type-checking 339
16.3 Translation of polymorphic programs 344
16.4 Resolution of static overloading 347

17 Dataflow Analysis 350
17.1 Intermediate representation for flow analysis 351
17.2 Various dataflow analyses 354
17.3 Transformations using dataflow analysis 359
17.4 Speeding up dataflow analysis 360
17.5 Alias analysis 369

18 Loop Optimizations 376
18.1 Dominators 379
18.2 Loop-invariant computations 384

18.3 Induction variables 385
18.4 Array-bounds checks 391
18.5 Loop unrolling 395

19 Static Single-Assignment Form 399
19.1 Converting to SSA form 402
19.2 Efficient computation of the dominator tree 410
19.3 Optimization algorithms using SSA 417
19.4 Arrays, pointers, and memory 423
19.5 The control-dependence graph 425
19.6 Converting back from SSA form 428
19.7 A functional intermediate form 430

20 Pipelining and Scheduling 440
20.1 Loop scheduling without resource bounds 444
20.2 Resource-bounded loop pipelining 448
20.3 Branch prediction 456

21 The Memory Hierarchy 464
21.1 Cache organization 465
21.2 Cache-block alignment 468
21.3 Prefetching 470
21.4 Loop interchange 476
21.5 Blocking 477
21.6 Garbage collection and the memory hierarchy 480

Appendix: MiniJava Language Reference Manual 484
A.1 Lexical Issues 484
A.2 Grammar 484
A.3 Sample Program 486

Bibliography 487

Index 495

Preface

This book is intended as a textbook for a one- or two-semester course in compilers. Students will see the theory behind different components of a compiler, the programming techniques used to put the theory into practice, and the interfaces used to modularize the compiler. To make the interfaces and programming examples clear and concrete, we have written them in Java. Another edition of this book is available that uses the ML language.

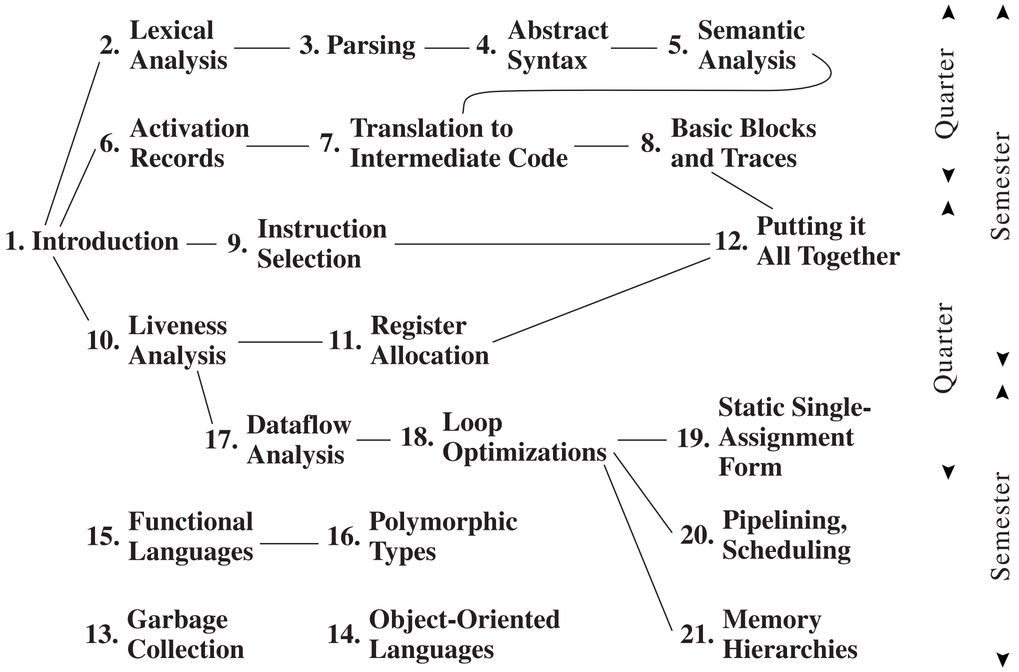
Implementation project. The "student project compiler" that we have outlined is reasonably simple, but is organized to demonstrate some important techniques that are now in common use: abstract syntax trees to avoid tangling syntax and semantics, separation of instruction selection from register allocation, copy propagation to give flexibility to earlier phases of the compiler, and containment of target-machine dependencies. Unlike many "student compilers" found in other textbooks, this one has a simple but sophisticated back end, allowing good register allocation to be done after instruction selection.

This second edition of the book has a redesigned project compiler: It uses a subset of Java, called MiniJava, as the source language for the compiler project, it explains the use of the parser generators JavaCC and SableCC, and it promotes programming with the Visitor pattern. Students using this edition can implement a compiler for a language they're familiar with, using standard tools, in a more object-oriented style.

Each chapter in Part I has a programming exercise corresponding to one module of a compiler. Software useful for the exercises can be found at
http://uk.cambridge.org/resources/052182060X (outside North America);
http://us.cambridge.org/titles/052182060X.html (within North America).

Exercises. Each chapter has pencil-and-paper exercises; those marked with a star are more challenging, two-star problems are difficult but solvable, and the occasional three-star exercises are not known to have a solution.

Course sequence. The figure shows how the chapters depend on each other.



- A one-semester course could cover all of Part I (Chapters 1–12), with students implementing the project compiler (perhaps working in groups); in addition, lectures could cover selected topics from Part II.
- An advanced or graduate course could cover Part II, as well as additional topics from the current literature. Many of the Part II chapters can stand independently from Part I, so that an advanced course could be taught to students who have used a different book for their first course.
- In a two-quarter sequence, the first quarter could cover Chapters 1–8, and the second quarter could cover Chapters 9–12 and some chapters from Part II.

Acknowledgments. Many people have provided constructive criticism or helped us in other ways on this book. Vidyut Samanta helped tremendously with both the text and the software for the new edition of the book. We would also like to thank Leonor Abraido-Fandino, Scott Ananian, Nils Andersen, Stephen Bailey, Joao Cangussu, Maia Ginsburg, Max Hailperin, David Hanson, Jeffrey Hsu, David MacQueen, Torben Mogensen, Doug Morgan, Robert Netzer, Elma Lee Noah, Mikael Petterson, Benjamin Pierce, Todd Proebsting, Anne Rogers, Barbara Ryder, Amr Sabry, Mooly Sagiv, Zhong Shao, Mary Lou Soffa, Andrew Tolmach, Kwangkeun Yi, and Kenneth Zadeck.

PART ONE

Fundamentals of Compilation

1 Introduction

A **compiler** was originally a program that "compiled" subroutines [a link-loader]. When in 1954 the combination "algebraic compiler" came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

Bauer and Eickel [1975]

This book describes techniques, data structures, and algorithms for translating programming languages into executable code. A modern compiler is often organized into many phases, each operating on a different abstract "language." The chapters of this book follow the organization of a compiler, each covering a successive phase.

To illustrate the issues in compiling real programming languages, we show how to compile MiniJava, a simple but nontrivial subset of Java. Programming exercises in each chapter call for the implementation of the corresponding phase; a student who implements all the phases described in Part I of the book will have a working compiler. MiniJava is easily extended to support class extension or higher-order functions, and exercises in Part II show how to do this. Other chapters in Part II cover advanced techniques in program optimization. Appendix A describes the MiniJava language.

The interfaces between modules of the compiler are almost as important as the algorithms inside the modules. To describe the interfaces concretely, it is useful to write them down in a real programming language. This book uses Java – a simple object-oriented language. Java is *safe*, in that programs cannot circumvent the type system to violate abstractions; and it has garbage collection, which greatly simplifies the management of dynamic storage al-

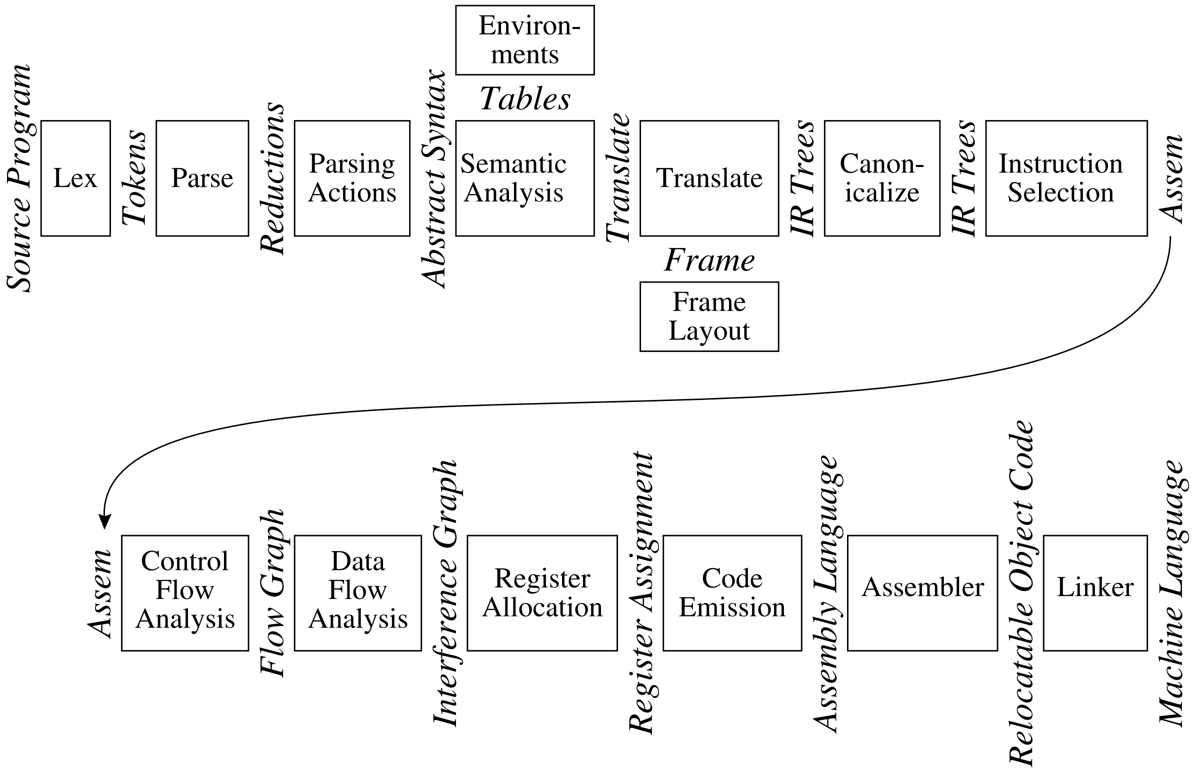


FIGURE 1.1. Phases of a compiler, and interfaces between them.

location. Both of these properties are useful in writing compilers (and almost any kind of software).

This is not a textbook on Java programming. Students using this book who do not know Java already should pick it up as they go along, using a Java programming book as a reference. Java is a small enough language, with simple enough concepts, that this should not be difficult for students with good programming skills in other languages.

1.1 MODULES AND INTERFACES

Any large software system is much easier to understand and implement if the designer takes care with the fundamental abstractions and interfaces. Figure 1.1 shows the phases in a typical compiler. Each phase is implemented as one or more software modules.

Breaking the compiler into this many pieces allows for reuse of the components. For example, to change the target machine for which the compiler produces machine language, it suffices to replace just the Frame Layout and Instruction Selection modules. To change the source language being compiled, only the modules up through Translate need to be changed. The compiler can be attached to a language-oriented syntax editor at the *Abstract Syntax* interface.

The learning experience of coming to the right abstraction by several iterations of *think–implement–redesign* is one that should not be missed. However, the student trying to finish a compiler project in one semester does not have this luxury. Therefore, we present in this book the outline of a project where the abstractions and interfaces are carefully thought out, and are as elegant and general as we are able to make them.

Some of the interfaces, such as *Abstract Syntax, IR Trees,* and *Assem,* take the form of data structures: For example, the Parsing Actions phase builds an *Abstract Syntax* data structure and passes it to the Semantic Analysis phase. Other interfaces are abstract data types; the *Translate* interface is a set of functions that the Semantic Analysis phase can call, and the *Tokens* interface takes the form of a function that the Parser calls to get the next token of the input program.

DESCRIPTION OF THE PHASES

Each chapter of Part I of this book describes one compiler phase, as shown in Table 1.2

This modularization is typical of many real compilers. But some compilers combine Parse, Semantic Analysis, Translate, and Canonicalize into one phase; others put Instruction Selection much later than we have done, and combine it with Code Emission. Simple compilers omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases.

We have designed the compiler in this book to be as simple as possible, but no simpler. In particular, in those places where corners are cut to simplify the implementation, the structure of the compiler allows for the addition of more optimization or fancier semantics without violence to the existing interfaces.

1.2 TOOLS AND SOFTWARE

Two of the most useful abstractions used in modern compilers are *context-free grammars*, for parsing, and *regular expressions*, for lexical analysis. To make the best use of these abstractions it is helpful to have special tools,

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or *tokens*.
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of *abstract syntax tree* corresponding to each phrase.
5	Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
7	Translate	Produce *intermediate representation trees* (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
8	Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a *control flow graph* that shows all the possible flows of control the program might follow when it executes.
10	Dataflow Analysis	Gather information about the flow of information through variables of the program; for example, *liveness analysis* calculates the places where each program variable holds a still-needed value (is *live*).
11	Register Allocation	Choose a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
12	Code Emission	Replace the temporary names in each machine instruction with machine registers.

TABLE 1.2. Description of compiler phases.

such as *Yacc* (which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical-analysis program). Fortunately, such tools are available for Java, and the project described in this book makes use of them.

The programming projects in this book can be compiled using any Java

Stm → *Stm* ; *Stm*	(CompoundStm)
Stm → `id` `:=` *Exp*	(AssignStm)
Stm → `print` (*ExpList*)	(PrintStm)
Exp → `id`	(IdExp)
Exp → `num`	(NumExp)
Exp → *Exp Binop Exp*	(OpExp)
Exp → (*Stm* , *Exp*)	(EseqExp)
ExpList → *Exp* , *ExpList*	(PairExpList)
ExpList → *Exp*	(LastExpList)
Binop → +	(Plus)
Binop → −	(Minus)
Binop → ×	(Times)
Binop → /	(Div)

GRAMMAR 1.3. A straight-line programming language.

compiler. The parser generators *JavaCC* and *SableCC* are freely available on the Internet; for information see the World Wide Web page

http://uk.cambridge.org/resources/052182060X (outside North America);
http://us.cambridge.org/titles/052182060X.html (within North America).

Source code for some modules of the MiniJava compiler, skeleton source code and support code for some of the programming exercises, example MiniJava programs, and other useful files are also available from the same Web address. The programming exercises in this book refer to this directory as `$MINIJAVA/` when referring to specific subdirectories and files contained therein.

1.3 DATA STRUCTURES FOR TREE LANGUAGES

Many of the important data structures used in a compiler are *intermediate representations* of the program being compiled. Often these representations take the form of trees, with several node types, each of which has different attributes. Such trees can occur at many of the phase-interfaces shown in Figure 1.1.

Tree representations can be described with grammars, just like programming languages. To introduce the concepts, we will show a simple programming language with statements and expressions, but no loops or if-statements (this is called a language of *straight-line programs*).

The syntax for this language is given in Grammar 1.3.

The informal semantics of the language is as follows. Each *Stm* is a statement, each *Exp* is an expression. $s_1; s_2$ executes statement s_1, then statement s_2. $i := e$ evaluates the expression e, then "stores" the result in variable i.

`print`($e_1, e_2, \ldots, e_n$) displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

An *identifier expression*, such as i, yields the current contents of the variable i. A *number* evaluates to the named integer. An *operator expression* e_1 op e_2 evaluates e_1, then e_2, then applies the given binary operator. And an *expression sequence* (s, e) behaves like the C-language "comma" operator, evaluating the statement s for side effects before evaluating (and returning the result of) the expression e.

For example, executing this program

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

prints

```
8 7
80
```

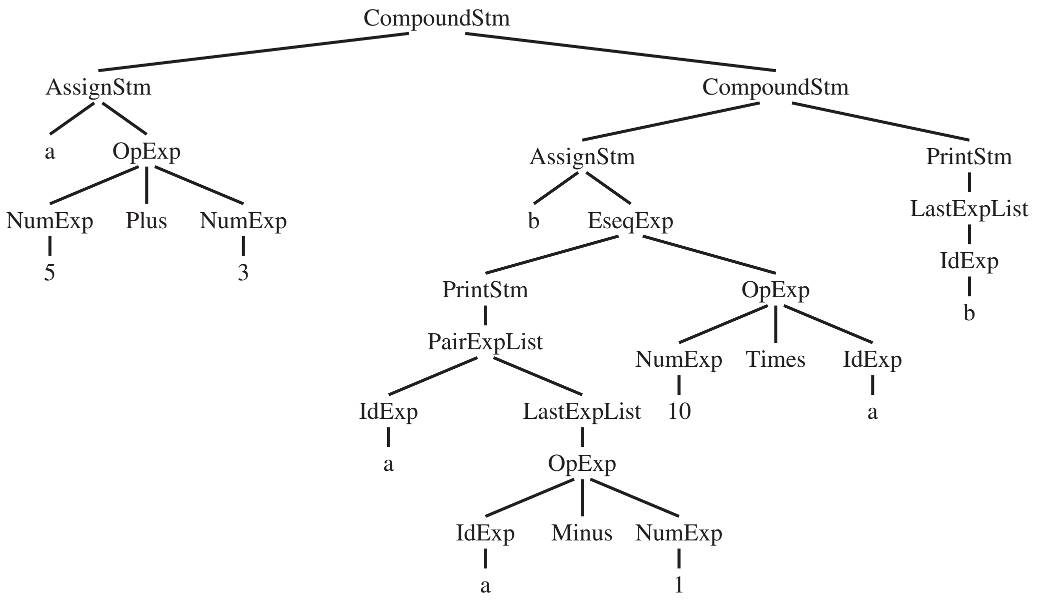
How should this program be represented inside a compiler? One representation is *source code*, the characters that the programmer writes. But that is not so easy to manipulate. More convenient is a tree data structure, with one node for each statement (Stm) and expression (Exp). Figure 1.4 shows a tree representation of the program; the nodes are labeled by the production labels of Grammar 1.3, and each node has as many children as the corresponding grammar production has right-hand-side symbols.

We can translate the grammar directly into data structure definitions, as shown in Program 1.5. Each grammar symbol corresponds to an `abstract class` in the data structures:

Grammar	class
Stm	Stm
Exp	Exp
ExpList	ExpList
id	String
num	int

For each grammar rule, there is one *constructor* that belongs to the class for its left-hand-side symbol. We simply *extend* the abstract class with a "concrete" class for each grammar rule. The constructor (class) names are indicated on the right-hand side of Grammar 1.3.

Each grammar rule has right-hand-side components that must be represented in the data structures. The CompoundStm has two Stm's on the right-hand side; the AssignStm has an identifier and an expression; and so on.



```
a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
```

FIGURE 1.4. Tree representation of a straight-line program.

These become *fields* of the subclasses in the Java data structure. Thus, CompoundStm has two fields (also called *instance variables*) called `stm1` and `stm2`; AssignStm has fields `id` and `exp`.

For Binop we do something simpler. Although we could make a Binop class – with subclasses for Plus, Minus, Times, Div – this is overkill because none of the subclasses would need any fields. Instead we make an “enumeration” type (in Java, actually an integer) of constants (`final int` variables) local to the OpExp class.

Programming style. We will follow several conventions for representing tree data structures in Java:

1. Trees are described by a grammar.
2. A tree is described by one or more abstract classes, each corresponding to a symbol in the grammar.
3. Each abstract class is *extended* by one or more subclasses, one for each grammar rule.

```
public abstract class Stm {}

public class CompoundStm extends Stm {
   public Stm stm1, stm2;
   public CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}}

public class AssignStm extends Stm {
   public String id; public Exp exp;
   public AssignStm(String i, Exp e) {id=i; exp=e;}}

public class PrintStm extends Stm {
   public ExpList exps;
   public PrintStm(ExpList e) {exps=e;}}

public abstract class Exp {}

public class IdExp extends Exp {
   public String id;
   public IdExp(String i) {id=i;}}

public class NumExp extends Exp {
   public int num;
   public NumExp(int n) {num=n;}}

public class OpExp extends Exp {
   public Exp left, right; public int oper;
   final public static int Plus=1,Minus=2,Times=3,Div=4;
   public OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}}

public class EseqExp extends Exp {
   public Stm stm; public Exp exp;
   public EseqExp(Stm s, Exp e) {stm=s; exp=e;}}

public abstract class ExpList {}

public class PairExpList extends ExpList {
   public Exp head; public ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}}

public class LastExpList extends ExpList {
   public Exp head;
   public LastExpList(Exp h) {head=h;}}
```

PROGRAM 1.5. Representation of straight-line programs.

4. For each nontrivial symbol in the right-hand side of a rule, there will be one field in the corresponding class. (A trivial symbol is a punctuation symbol such as the semicolon in CompoundStm.)
5. Every class will have a constructor function that initializes all the fields.
6. Data structures are initialized when they are created (by the constructor functions), and are never modified after that (until they are eventually discarded).

Modularity principles for Java programs. A compiler can be a big program; careful attention to modules and interfaces prevents chaos. We will use these principles in writing a compiler in Java:

1. Each phase or module of the compiler belongs in its own package.
2. "Import on demand" declarations will not be used. If a Java file begins with

   ```
   import A.F.*; import A.G.*; import B.*; import C.*;
   ```

   then the human reader *will have to look outside this file* to tell which package defines the `X` that is used in the expression `X.put()`.
3. "Single-type import" declarations are a better solution. If the module begins

   ```
   import A.F.W; import A.G.X; import B.Y; import C.Z;
   ```

   then you can tell *without looking outside this file* that `X` comes from `A.G`.
4. Java is naturally a multithreaded system. We would like to support multiple simultaneous compiler threads and compile two different programs simultaneously, one in each compiler thread. Therefore, static variables must be avoided unless they are `final` (constant). We never want two compiler threads to be updating the same (static) instance of a variable.

PROGRAM STRAIGHT-LINE PROGRAM INTERPRETER

Implement a simple program analyzer and interpreter for the straight-line programming language. This exercise serves as an introduction to *environments* (symbol tables mapping variable names to information about the variables); to *abstract syntax* (data structures representing the phrase structure of programs); to *recursion over tree data structures*, useful in many parts of a compiler; and to a *functional style* of programming without assignment statements.

It also serves as a "warm-up" exercise in Java programming. Programmers experienced in other languages but new to Java should be able to do this exercise, but will need supplementary material (such as textbooks) on Java.

Programs to be interpreted are already parsed into abstract syntax, as described by the data types in Program 1.5.

However, we do not wish to worry about parsing the language, so we write this program by applying data constructors:

```
Stm prog =
new CompoundStm(new AssignStm("a",
                     new OpExp(new NumExp(5),
                               OpExp.Plus, new NumExp(3))),
 new CompoundStm(new AssignStm("b",
    new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
              new LastExpList(new OpExp(new IdExp("a"),
                            OpExp.Minus,new NumExp(1))))),

          new OpExp(new NumExp(10), OpExp.Times,
                    new IdExp("a")))),
  new PrintStm(new LastExpList(new IdExp("b")))));
```

Files with the data type declarations for the trees, and this sample program, are available in the directory `$MINIJAVA/chap1`.

Writing interpreters without side effects (that is, assignment statements that update variables and data structures) is a good introduction to *denotational semantics* and *attribute grammars*, which are methods for describing what programming languages do. It's often a useful technique in writing compilers, too; compilers are also in the business of saying what programming languages do.

Therefore, in implementing these programs, never assign a new value to any variable or object field except when it is initialized. For local variables, use the initializing form of declaration (for example, `int i=j+3;`) and for each class, make a constructor function (like the `CompoundStm` constructor in Program 1.5).

1. Write a Java function `int maxargs(Stm s)` that tells the maximum number of arguments of any `print` statement within any subexpression of a given statement. For example, `maxargs(prog)` is 2.

2. Write a Java function `void interp(Stm s)` that "interprets" a program in this language. To write in a "functional programming" style – in which you never use an assignment statement – initialize each local variable as you declare it.

Your functions that examine each `Exp` will have to use `instanceof` to determine which subclass the expression belongs to and then cast to the proper subclass. Or you can add methods to the `Exp` and `Stm` classes to avoid the use of `instanceof`.

For part 1, remember that print statements can contain expressions that contain other print statements.

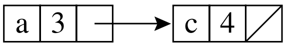
For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a "table," mapping identifiers to the integer values assigned to them, as a list of `id` × `int` pairs.

```
class Table {
  String id; int value; Table tail;
  Table(String i, int v, Table t) {id=i; value=v; tail=t;}
}
```

Then `interpStm` is declared as

```
Table interpStm(Stm s, Table t)
```

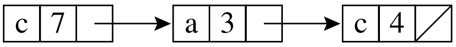
taking a table t_1 as argument and producing the new table t_2 that's just like t_1 except that some identifiers map to different integers as a result of the statement.

For example, the table t_1 that maps a to 3 and maps c to 4, which we write $\{a \mapsto 3, c \mapsto 4\}$ in mathematical notation, could be represented as the linked list  .

Now, let the table t_2 be just like t_1, except that it maps c to 7 instead of 4. Mathematically, we could write,

$$t_2 = \text{update}(t_1, c, 7),$$

where the update function returns a new table $\{a \mapsto 3, c \mapsto 7\}$.

On the computer, we could implement t_2 by putting a new cell at the head of the linked list: , as long as we assume that the *first* occurrence of c in the list takes precedence over any later occurrence.

Therefore, the `update` function is easy to implement; and the corresponding `lookup` function

```
int lookup(Table t, String key)
```

just searches down the linked list. Of course, in an object-oriented style, `int lookup(String key)` should be a method of the `Table` class.

Interpreting expressions is more complicated than interpreting statements, because expressions return integer values *and* have side effects. We wish to simulate the straight-line programming language's assignment statements without doing any side effects in the interpreter itself. (The `print` statements will be accomplished by interpreter side effects, however.) The solution is to declare `interpExp` as

```
class IntAndTable {int i; Table t;
    IntAndTable(int ii, Table tt) {i=ii; t=tt;}
   }
IntAndTable interpExp(Exp e, Table t) ···
```

The result of interpreting an expression e_1 with table t_1 is an integer value i and a new table t_2. When interpreting an expression with two subexpressions (such as an `OpExp`), the table t_2 resulting from the first subexpression can be used in processing the second subexpression.

EXERCISES

1.1 This simple program implements *persistent* functional binary search trees, so that if `tree2=insert(x,tree1)`, then `tree1` is still available for lookups even while `tree2` can be used.

```
class Tree {Tree left; String key; Tree right;
   Tree(Tree l, String k, Tree r) {left=l; key=k; right=r;}

Tree insert(String key, Tree t) {
  if (t==null) return new Tree(null, key, null)
  else if (key.compareTo(t.key) < 0)
        return new Tree(insert(key,t.left),t.key,t.right);
  else if (key.compareTo(t.key) > 0)
        return new Tree(t.left,t.key,insert(key,t.right));
  else return new Tree(t.left,key,t.right);
}
```

a. Implement a `member` function that returns `true` if the item is found, else `false`.

b. Extend the program to include not just membership, but the mapping of keys to bindings:

```
Tree insert(String key, Object binding, Tree t);
Object lookup(String key, Tree t);
```

c. These trees are not balanced; demonstrate the behavior on the following two sequences of insertions:
(a) t s p i p f b s t
(b) a b c d e f g h i

*d. Research balanced search trees in Sedgewick [1997] and recommend a balanced-tree data structure for functional symbol tables. **Hint:** To preserve a functional style, the algorithm should be one that rebalances on insertion but not on lookup, so a data structure such as *splay trees* is not appropriate.

e. Rewrite in an object-oriented (but still "functional") style, so that insertion is now `t.insert(key)` instead of `insert(key,t)`. **Hint:** You'll need an `EmptyTree` subclass.

2 Lexical Analysis

lex-i-cal: of or relating to words or the vocabulary of a language as distinguished from its grammar and construction

Webster's Dictionary

To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way. The front end of the compiler performs analysis; the back end does synthesis.

The analysis is usually broken up into

Lexical analysis: breaking the input into individual words or “tokens”;
Syntax analysis: parsing the phrase structure of the program; and
Semantic analysis: calculating the program’s meaning.

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens. It would unduly complicate the parser to have to account for possible white space and comments at every possible point; this is the main reason for separating lexical analysis from parsing.

Lexical analysis is not very complicated, but we will attack it with high-powered formalisms and tools, because similar formalisms will be useful in the study of parsing and similar tools have many applications in areas other than compilation.

2.1 LEXICAL TOKENS

A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language. A programming language classifies lexical tokens into a finite set of token types. For example, some of the token types of a typical programming language are

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Punctuation tokens such as IF, VOID, RETURN constructed from alphabetic characters are called *reserved words* and, in most languages, cannot be used as identifiers.

Examples of nontokens are

comment	`/* try again */`
preprocessor directive	`#include<stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, and newlines	

In languages weak enough to require a macro preprocessor, the preprocessor operates on the source character stream, producing another character stream that is then fed to the lexical analyzer. It is also possible to integrate macro processing with lexical analysis.

Given a program such as

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

the lexical analyzer will return the stream

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s)

```
COMMA    STRING(0.0)    COMMA    NUM(3)    RPAREN    RPAREN
RETURN   REAL(0.0)    SEMI    RBRACE    EOF
```

where the token-type of each token is reported; some of the tokens, such as identifiers and literals, have *semantic values* attached to them, giving auxiliary information in addition to the token-type.

How should the lexical rules of a programming language be described? In what language should a lexical analyzer be written?

We can describe the lexical tokens of a language in English; here is a description of identifiers in C or Java:

> An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

And any reasonable programming language serves to implement an ad hoc lexer. But we will specify lexical tokens using the formal language of *regular expressions*, implement lexers using *deterministic finite automata*, and use mathematics to connect the two. This will lead to simpler and more readable lexical analyzers.

2.2 REGULAR EXPRESSIONS

Let us say that a *language* is a set of *strings*; a string is a finite sequence of *symbols*. The symbols themselves are taken from a finite *alphabet*.

The Pascal language is the set of all strings that constitute legal Pascal programs; the language of primes is the set of all decimal-digit strings that represent prime numbers; and the language of C reserved words is the set of all alphabetic strings that cannot be used as identifiers in the C programming language. The first two of these languages are infinite sets; the last is a finite set. In all of these cases, the alphabet is the ASCII character set.

When we speak of languages in this way, we will not assign any meaning to the strings; we will just be attempting to classify each string as in the language or not.

To specify some of these (possibly infinite) languages with finite descriptions, we will use the notation of *regular expressions*. Each regular expression stands for a set of strings.

Symbol: For each symbol **a** in the alphabet of the language, the regular expression **a** denotes the language containing just the string `a`.

Alternation: Given two regular expressions M and N, the alternation operator written as a vertical bar | makes a new regular expression $M \mid N$. A string is in the language of $M \mid N$ if it is in the language of M or in the language of N. Thus, the language of $\mathbf{a} \mid \mathbf{b}$ contains the two strings `a` and `b`.

Concatenation: Given two regular expressions M and N, the concatenation operator · makes a new regular expression $M \cdot N$. A string is in the language of $M \cdot N$ if it is the concatenation of any two strings α and β such that α is in the language of M and β is in the language of N. Thus, the regular expression $(\mathbf{a} \mid \mathbf{b}) \cdot \mathbf{a}$ defines the language containing the two strings `aa` and `ba`.

Epsilon: The regular expression ϵ represents a language whose only string is the empty string. Thus, $(a \cdot b) \mid \epsilon$ represents the language {`""`,`"ab"`}.

Repetition: Given a regular expression M, its Kleene closure is M^*. A string is in M^* if it is the concatenation of zero or more strings, all of which are in M. Thus, $((\mathbf{a} \mid \mathbf{b}) \cdot \mathbf{a})^*$ represents the infinite set { `""` , `"aa"`, `"ba"`, `"aaaa"`, `"baaa"`, `"aaba"`, `"baba"`, `"aaaaaa"`, ... }.

Using symbols, alternation, concatenation, epsilon, and Kleene closure we can specify the set of ASCII characters corresponding to the lexical tokens of a programming language. First, consider some examples:

$(\mathbf{0} \mid \mathbf{1})^* \cdot \mathbf{0}$	Binary numbers that are multiples of two.
$\mathbf{b}^*(\mathbf{abb}^*)^*(\mathbf{a}\mid\epsilon)$	Strings of `a`'s and `b`'s with no consecutive `a`'s.
$(\mathbf{a}\mid\mathbf{b})^*\mathbf{aa}(\mathbf{a}\mid\mathbf{b})^*$	Strings of `a`'s and `b`'s containing consecutive `a`'s.

In writing regular expressions, we will sometimes omit the concatenation symbol or the epsilon, and we will assume that Kleene closure "binds tighter" than concatenation, and concatenation binds tighter than alternation; so that $\mathbf{ab} \mid \mathbf{c}$ means $(\mathbf{a} \cdot \mathbf{b}) \mid \mathbf{c}$, and $(\mathbf{a} \mid)$ means $(\mathbf{a} \mid \epsilon)$.

Let us introduce some more abbreviations: [**abcd**] means $(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \mathbf{d})$, [**b**-**g**] means [**bcdefg**], [**b**-**gM**-**Qkr**] means [**bcdefgMNOPQkr**], $M?$ means $(M \mid \epsilon)$, and M^+ means $(M \cdot M^*)$. These extensions are convenient, but none extend the descriptive power of regular expressions: Any set of strings that can be described with these abbreviations could also be described by just the basic set of operators. All the operators are summarized in Figure 2.1.

Using this language, we can specify the lexical tokens of a programming language (Figure 2.2).

The fifth line of the description recognizes comments or white space but

a	An ordinary character stands for itself.
ϵ	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to write concatenation.
M^*	Repetition (zero or more times).
M^+	Repetition, one or more times.
$M?$	Optional, zero or one occurrence of M.
$[\mathbf{a-zA-Z}]$	Character set alternation.
.	A period stands for any single character except newline.
`"a.+*"`	Quotation, a string in quotes stands for itself literally.

FIGURE 2.1. Regular expression notation.

`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

FIGURE 2.2. Regular expressions for some tokens.

does not report back to the parser. Instead, the white space is discarded and the lexer resumed. The comments for this lexer begin with two dashes, contain only alphabetic characters, and end with newline.

Finally, a lexical specification should be *complete*, always matching some initial substring of the input; we can always achieve this by having a rule that matches any single character (and in this case, prints an "illegal character" error message and continues).

These rules are a bit ambiguous. For example, does `if8` match as a single identifier or as the two tokens `if` and `8`? Does the string `if 89` begin with an identifier or a reserved word? There are two important disambiguation rules used by Lex, JavaCC, SableCC, and other similar lexical-analyzer generators:

Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule priority: For a *particular* longest initial substring, the first regular expression that can match determines its token-type. This means that the order of

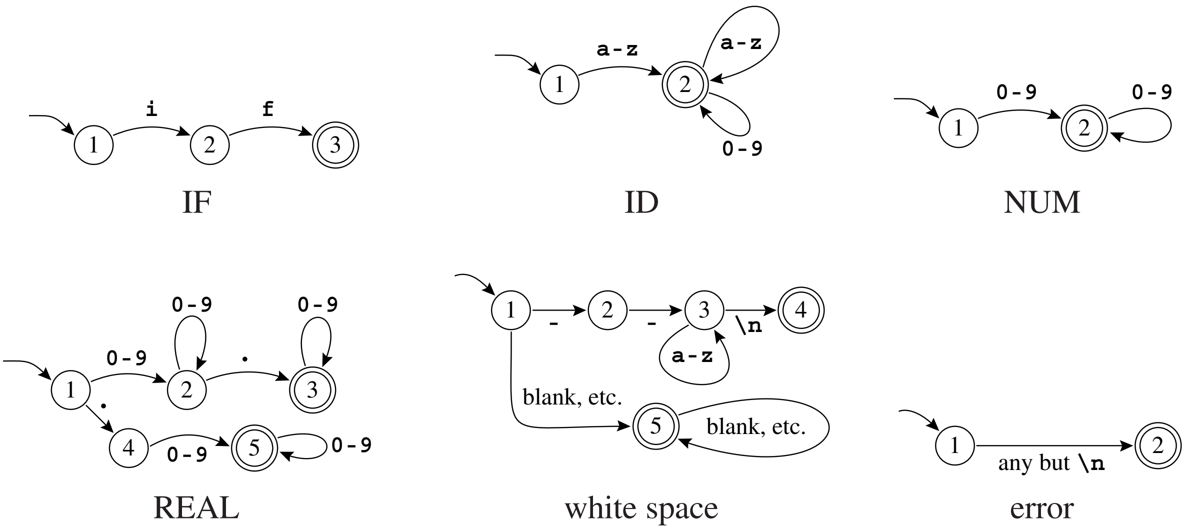


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

writing down the regular-expression rules has significance.

Thus, `if8` matches as an identifier by the longest-match rule, and `if` matches as a reserved word by rule-priority.

2.3 FINITE AUTOMATA

Regular expressions are convenient for specifying lexical tokens, but we need a formalism that can be implemented as a computer program. For this we can use finite automata (N.B. the singular of automata is automaton). A finite automaton has a finite set of *states*; *edges* lead from one state to another, and each edge is labeled with a *symbol*. One state is the *start* state, and certain of the states are distinguished as *final* states.

Figure 2.3 shows some finite automata. We number the states just for convenience in discussion. The start state is numbered 1 in each case. An edge labeled with several characters is shorthand for many parallel edges; so in the ID machine there are really 26 edges each leading from state 1 to 2, each labeled by a different letter.

In a *deterministic* finite automaton (DFA), no two edges leaving from the

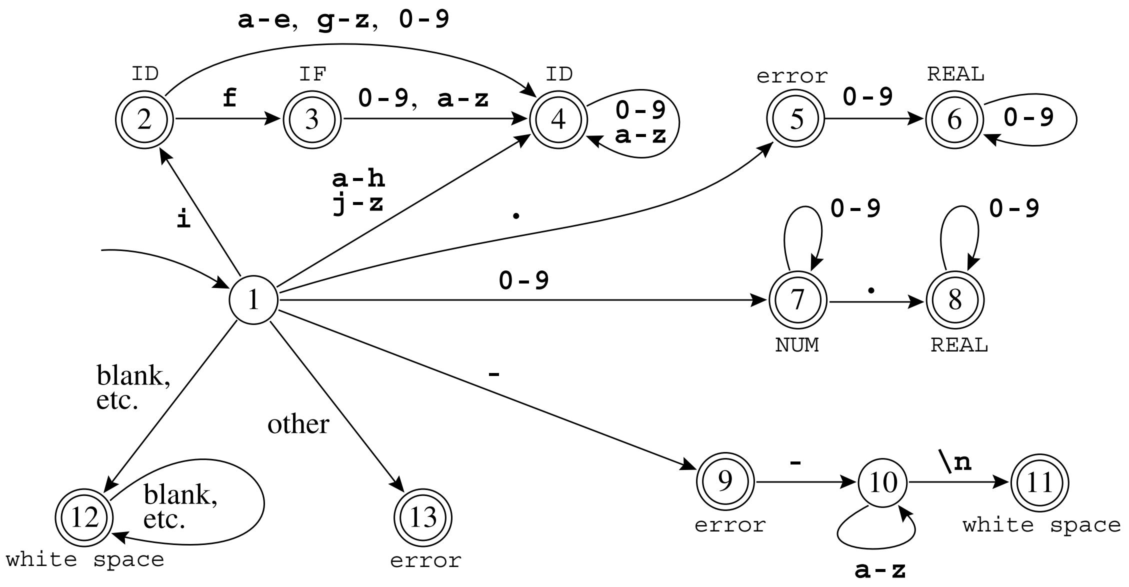


FIGURE 2.4. Combined finite automaton.

same state are labeled with the same symbol. A DFA *accepts* or *rejects* a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character. After making *n* transitions for an *n*-character string, if the automaton is in a final state, then it accepts the string. If it is not in a final state, or if at some point there was no appropriately labeled edge to follow, it rejects. The *language* recognized by an automaton is the set of strings that it accepts.

For example, it is clear that any string in the language recognized by automaton ID must begin with a letter. Any single letter leads to state 2, which is final; so a single-letter string is accepted. From state 2, any letter or digit leads back to state 2, so a letter followed by any number of letters and digits is also accepted.

In fact, the machines shown in Figure 2.3 accept the same languages as the regular expressions of Figure 2.2.

These are six separate automata; how can they be combined into a single machine that can serve as a lexical analyzer? We will study formal ways of doing this in the next section, but here we will just do it ad hoc: Figure 2.4 shows such a machine. Each final state must be labeled with the token-type that it accepts. State 2 in this machine has aspects of state 2 of the IF machine and state 2 of the ID machine; since the latter is final, then the combined state must be final. State 3 is like state 3 of the IF machine and state 2 of the ID machine; because these are both final we use *rule priority* to disambiguate – we label state 3 with IF because we want this token to be recognized as a reserved word, not an identifier.

We can encode this machine as a transition matrix: a two-dimensional array (a vector of vectors), subscripted by state number and input character. There will be a “dead” state (state 0) that loops to itself on all characters; we use this to encode the absence of an edge.

```
int edges[][] = {  /* ···0 1 2···-···e f g h i j··· */
/* state 0 */    {0,0,···0,0,0···0···0,0,0,0,0,0···},
/* state 1 */    {0,0,···7,7,7···9···4,4,4,4,2,4···},
/* state 2 */    {0,0,···4,4,4···0···4,3,4,4,4,4···},
/* state 3 */    {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 4 */    {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 5 */    {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 6 */    {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 7 */    {0,0,···7,7,7···0···0,0,0,0,0,0···},
/* state 8 */    {0,0,···8,8,8···0···0,0,0,0,0,0···},
   et cetera
}
```

There must also be a “finality” array, mapping state numbers to actions – final state 2 maps to action ID, and so on.

RECOGNIZING THE LONGEST MATCH

It is easy to see how to use this table to recognize whether to accept or reject a string, but the job of a lexical analyzer is to find the longest match, the longest initial substring of the input that is a valid token. While interpreting transitions, the lexer must keep track of the longest match seen so far, and the position of that match.

Keeping track of the longest match just means remembering the last time the automaton was in a final state with two variables, `Last-Final` (the state number of the most recent final state encountered) and `Input-Position-at-Last-Final`. Every time a final state is entered, the lexer updates these variables; when a *dead* state (a nonfinal state with no output transitions) is reached, the variables tell what token was matched, and where it ended.

Figure 2.5 shows the operation of a lexical analyzer that recognizes longest matches; note that the current input position may be far beyond the most recent position at which the recognizer was in a final state.

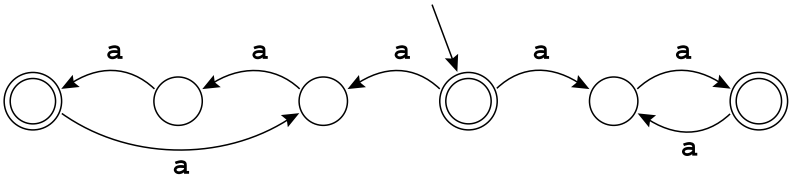
Last Final	Current State	Current Input	Accept Action
0	1	`|⊤⊥if --not-a-com`	
2	2	`|i⊤⊥f --not-a-com`	
3	3	`|if⊤⊥ --not-a-com`	
3	0	`|if⊤ ⊥--not-a-com`	*return* IF
0	1	`if|⊤⊥ --not-a-com`	
12	12	`if| ⊤⊥--not-a-com`	
12	0	`if| ⊤-⊥-not-a-com`	*found white space; resume*
0	1	`if |⊤⊥--not-a-com`	
9	9	`if |-⊤⊥-not-a-com`	
9	10	`if |-⊤-⊥not-a-com`	
9	10	`if |-⊤-n⊥ot-a-com`	
9	10	`if |-⊤-no⊥t-a-com`	
9	10	`if |-⊤-not⊥-a-com`	
9	0	`if |-⊤-not-⊥a-com`	*error, illegal token '-'; resume*
0	1	`if -|⊤⊥-not-a-com`	
9	9	`if -|-⊤⊥not-a-com`	
9	0	`if -|-⊤n⊥ot-a-com`	*error, illegal token '-'; resume*

FIGURE 2.5. The automaton of Figure 2.4 recognizes several tokens. The symbol | indicates the input position at each successive call to the lexical analyzer, the symbol ⊥ indicates the current position of the automaton, and ⊤ indicates the most recent position in which the recognizer was in a final state.

2.4 NONDETERMINISTIC FINITE AUTOMATA

A nondeterministic finite automaton (NFA) is one that has a choice of edges – labeled with the same symbol – to follow out of a state. Or it may have special edges labeled with ϵ (the Greek letter epsilon) that can be followed without eating any symbol from the input.

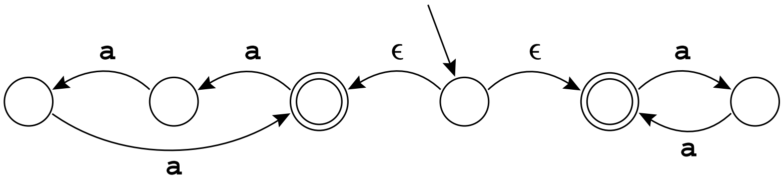
Here is an example of an NFA:



In the start state, on input character a, the automaton can move either right or left. If left is chosen, then strings of a's whose length is a multiple of three will be accepted. If right is chosen, then even-length strings will be accepted. Thus, the language recognized by this NFA is the set of all strings of a's whose length is a multiple of two or three.

On the first transition, this machine must choose which way to go. It is required to accept the string if there is *any* choice of paths that will lead to acceptance. Thus, it must "guess," and must always guess correctly.

Edges labeled with ϵ may be taken without using up a symbol from the input. Here is another NFA that accepts the same language:

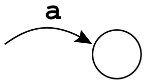


Again, the machine must choose which ϵ-edge to take. If there is a state with some ϵ-edges and some edges labeled by symbols, the machine can choose to eat an input symbol (and follow the corresponding symbol-labeled edge), or to follow an ϵ-edge instead.

CONVERTING A REGULAR EXPRESSION TO AN NFA

Nondeterministic automata are a useful notion because it is easy to convert a (static, declarative) regular expression to a (simulatable, quasi-executable) NFA.

The conversion algorithm turns each regular expression into an NFA with a *tail* (start edge) and a *head* (ending state). For example, the single-symbol regular expression **a** converts to the NFA



The regular expression **ab**, made by combining **a** with **b** using concatenation, is made by combining the two NFAs, hooking the head of **a** to the tail of **b**. The resulting machine has a tail labeled by **a** and a head into which the **b** edge flows.

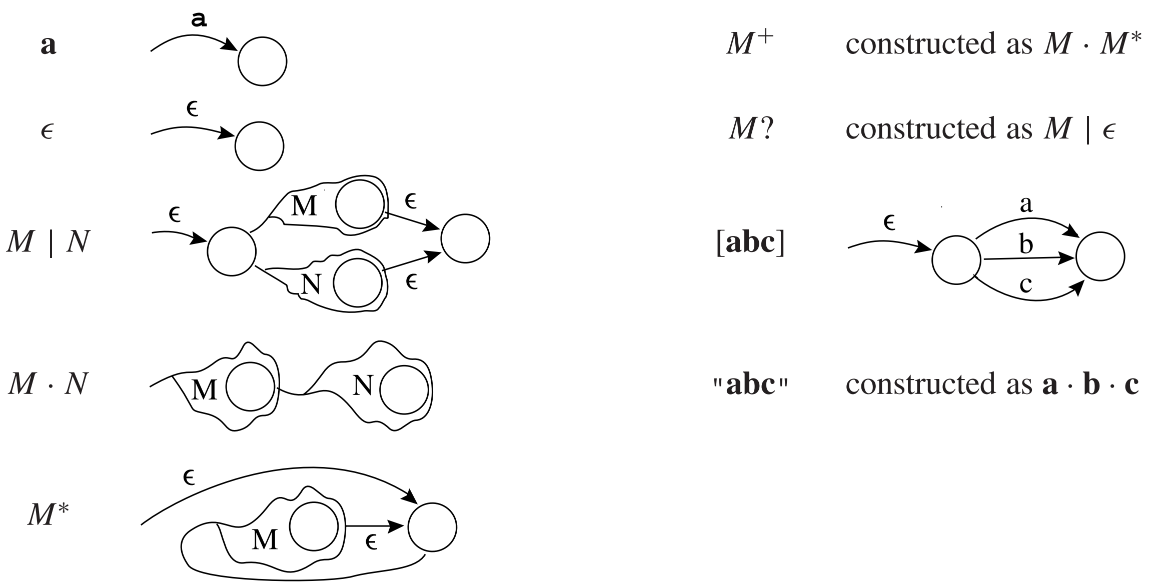
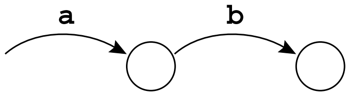
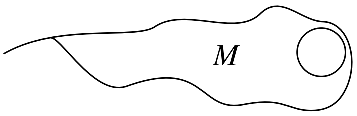


FIGURE 2.6. Translation of regular expressions to NFAs.



In general, any regular expression M will have some NFA with a tail and head:



We can define the translation of regular expressions to NFAs by induction. Either an expression is primitive (a single symbol or ϵ) or it is made from smaller expressions. Similarly, the NFA will be primitive or made from smaller NFAs.

Figure 2.6 shows the rules for translating regular expressions to nondeterministic automata. We illustrate the algorithm on some of the expressions in Figure 2.2 – for the tokens IF, ID, NUM, and **error**. Each expression is translated to an NFA, the "head" state of each NFA is marked final with a different token type, and the tails of all the expressions are joined to a new start node. The result – after some merging of equivalent NFA states – is shown in Figure 2.7.

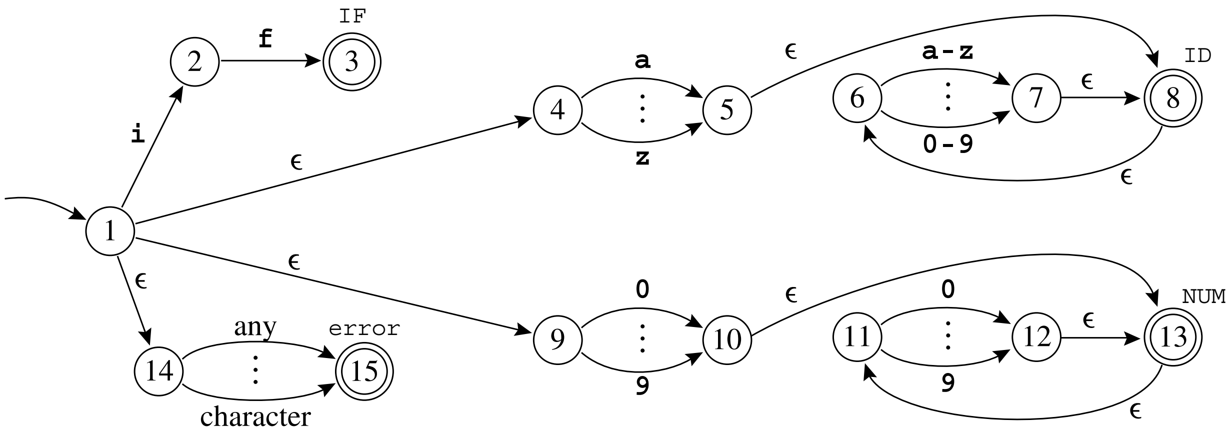


FIGURE 2.7. Four regular expressions translated to an NFA.

CONVERTING AN NFA TO A DFA

As we saw in Section 2.3, implementing deterministic finite automata (DFAs) as computer programs is easy. But implementing NFAs is a bit harder, since most computers don't have good "guessing" hardware.

We can avoid the need to guess by trying every possibility at once. Let us simulate the NFA of Figure 2.7 on the string `in`. We start in state 1. Now, instead of guessing which ϵ-transition to take, we just say that at this point the NFA might take any of them, so it is in one of the states $\{1, 4, 9, 14\}$; that is, we compute the *ϵ-closure* of $\{1\}$. Clearly, there are no other states reachable without eating the first character of the input.

Now, we make the transition on the character `i`. From state 1 we can reach 2, from 4 we reach 5, from 9 we go nowhere, and from 14 we reach 15. So we have the set $\{2, 5, 15\}$. But again we must compute the ϵ-closure: From 5 there is an ϵ-transition to 8, and from 8 to 6. So the NFA must be in one of the states $\{2, 5, 6, 8, 15\}$.

On the character `n`, we get from state 6 to 7, from 2 to nowhere, from 5 to nowhere, from 8 to nowhere, and from 15 to nowhere. So we have the set $\{7\}$; its ϵ-closure is $\{6, 7, 8\}$.

Now we are at the end of the string `in`; is the NFA in a final state? One of the states in our possible-states set is 8, which is final. Thus, `in` is an ID token.

We formally define ϵ-closure as follows. Let **edge**(s, c) be the set of all NFA states reachable by following a single edge with label c from state s.

For a set of states S, **closure**(S) is the set of states that can be reached from a state in S without consuming any of the input, that is, by going only through ϵ-edges. Mathematically, we can express the idea of going through ϵ-edges by saying that **closure**(S) is the smallest set T such that

$$T = S \cup \left(\bigcup_{s \in T} \mathbf{edge}(s, \epsilon) \right).$$

We can calculate T by iteration:

```
$T \leftarrow S$
**repeat** $T' \leftarrow T$
        $T \leftarrow T' \cup (\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon))$
**until** $T = T'$
```

Why does this algorithm work? T can only grow in each iteration, so the final T must include S. If $T = T'$ after an iteration step, then T must also include $\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon)$. Finally, the algorithm must terminate, because there are only a finite number of distinct states in the NFA.

Now, when simulating an NFA as described above, suppose we are in a set $d = \{s_i, s_k, s_l\}$ of NFA states s_i, s_k, s_l. By starting in d and eating the input symbol c, we reach a new set of NFA states; we'll call this set **DFAedge**(d, c):

$$\mathbf{DFAedge}(d, c) = \mathbf{closure}(\bigcup_{s \in d} \mathbf{edge}(s, c))$$

Using **DFAedge**, we can write the NFA simulation algorithm more formally. If the start state of the NFA is s_1, and the input string is $c_1, \ldots, c_k$, then the algorithm is

```
$d \leftarrow \mathbf{closure}(\{s_1\})$
**for** $i \leftarrow 1$ **to** $k$
  $d \leftarrow \mathbf{DFAedge}(d, c_i)$
```

Manipulating sets of states is expensive – too costly to want to do on every character in the source program that is being lexically analyzed. But it is possible to do all the sets-of-states calculations in advance. We make a DFA from the NFA, such that each set of NFA states corresponds to one DFA state. Since the NFA has a finite number n of states, the DFA will also have a finite number (at most 2^n) of states.

DFA construction is easy once we have **closure** and **DFAedge** algorithms. The DFA start state d_1 is just **closure**(s_1), as in the NFA simulation algo-

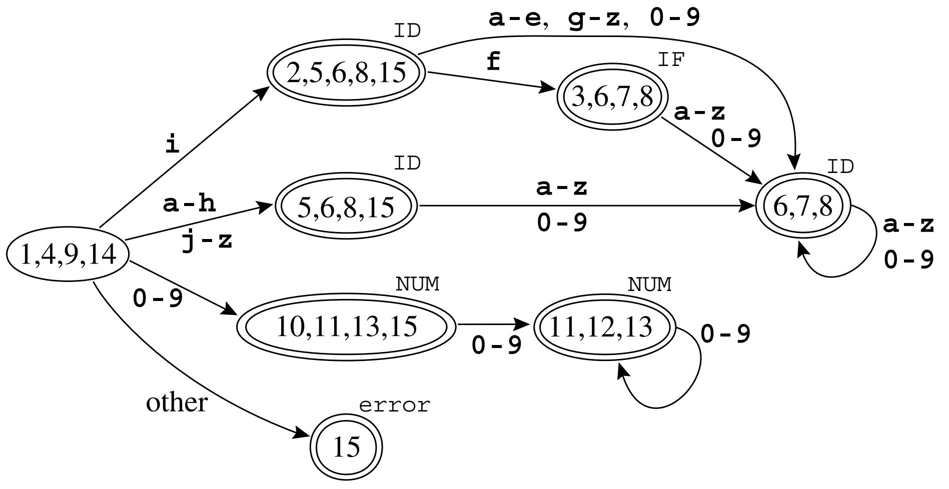


FIGURE 2.8. NFA converted to DFA.

rithm. Abstractly, there is an edge from d_i to d_j labeled with c if d_j = **DFAedge**(d_i, c). We let Σ be the alphabet.

```
states[0] ← {};      states[1] ← closure({$s_1$})
$p$ ← 1;      $j$ ← 0
while $j \le p$
  foreach $c \in \Sigma$
    $e$ ← DFAedge(states[$j$], $c$)
    if $e$ = states[$i$] for some $i \le p$
      then trans[$j, c$] ← $i$
      else $p \leftarrow p + 1$
           states[$p$] ← $e$
           trans[$j, c$] ← $p$
  $j \leftarrow j + 1$
```

The algorithm does not visit unreachable states of the DFA. This is extremely important, because in principle the DFA has 2^n states, but in practice we usually find that only about n of them are reachable from the start state. It is important to avoid an exponential blowup in the size of the DFA interpreter's transition tables, which will form part of the working compiler.

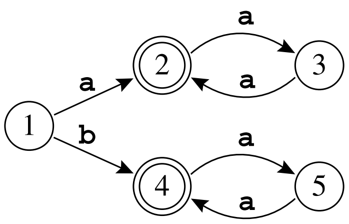
A state d is *final* in the DFA if any NFA state in states[d] is final in the NFA. Labeling a state *final* is not enough; we must also say what token is recognized; and perhaps several members of states[d] are final in the NFA. In this case we label d with the token-type that occurred first in the list of regular expressions that constitute the lexical specification. This is how *rule priority* is implemented.

After the DFA is constructed, the "states" array may be discarded, and the "trans" array is used for lexical analysis.

Applying the DFA construction algorithm to the NFA of Figure 2.7 gives the automaton in Figure 2.8.

This automaton is suboptimal. That is, it is not the smallest one that recognizes the same language. In general, we say that two states s_1 and s_2 are equivalent when the machine starting in s_1 accepts a string σ if and only if starting in s_2 it accepts σ. This is certainly true of the states labeled $\boxed{5,6,8,15}$ and $\boxed{6,7,8}$ in Figure 2.8, and of the states labeled $\boxed{10,11,13,15}$ and $\boxed{11,12,13}$. In an automaton with two equivalent states s_1 and s_2, we can make all of s_2's incoming edges point to s_1 instead and delete s_2.

How can we find equivalent states? Certainly, s_1 and s_2 are equivalent if they are both final or both nonfinal and, for any symbol c, $\text{trans}[s_1, c] = \text{trans}[s_2, c]$; $\boxed{10,11,13,15}$ and $\boxed{11,12,13}$ satisfy this criterion. But this condition is not sufficiently general; consider the automaton



Here, states 2 and 4 are equivalent, but $\text{trans}[2, a] \neq \text{trans}[4, a]$.

After constructing a DFA it is useful to apply an algorithm to minimize it by finding equivalent states; see Exercise 2.6.

2.5 LEXICAL-ANALYZER GENERATORS

DFA construction is a mechanical task easily performed by computer, so it makes sense to have an automatic *lexical-analyzer generator* to translate regular expressions into a DFA.

JavaCC and SableCC generate lexical analyzers and parsers written in Java. The lexical analyzers are generated from *lexical specifications*; and, as explained in the next chapter, the parsers are generated from grammars.

```
PARSER_BEGIN(MyParser)
   class MyParser {}
PARSER_END(MyParser)

/* For the regular expressions on the right, the token on the left will be returned: */
TOKEN : {
    < IF: "if" >
  | < #DIGIT: ["0"-"9"] >
  | < ID: ["a"-"z"] (["a"-"z"]|<DIGIT>)* >
  | < NUM: (<DIGIT>)+  >
  | < REAL: ( (<DIGIT>)+ "." (<DIGIT>)* ) |
            ( (<DIGIT>)* "." (<DIGIT>)+ )>
}

/* The regular expressions here will be skipped during lexical analysis: */
SKIP : {
    <"--" (["a"-"z"])* ("\n" | "\r" | "\r\n")>
  |  " "
  |  "\t"
  |  "\n"
}

/* If we have a substring that does not match any of the regular expressions in TOKEN or SKIP,
   JavaCC will automatically throw an error. */
void Start() :
{}
{  ( <IF> | <ID> | <NUM> | <REAL> )*  }
```

PROGRAM 2.9. JavaCC specification of the tokens from Figure 2.2.

For both JavaCC and SableCC, the lexical specification and the grammar are contained in the same file.

JAVACC

The tokens described in Figure 2.2 are specified in JavaCC as shown in Program 2.9. A JavaCC specification starts with an optional list of options followed by a Java compilation unit enclosed between `PARSER_BEGIN(name)` and `PARSER_END(name)`. The same name must follow `PARSER_BEGIN` and `PARSER_END`; it will be the name of the generated parser (`MyParser` in Program 2.9). The enclosed compilation unit must contain a class declaration of the same name as the generated parser.

Next is a list of grammar productions of the following kinds: a *regular-*

```
Helpers
    digit = [’0’..’9’];
Tokens
    if = ’if’;
    id = [’a’..’z’]([’a’..’z’] | (digit))*;
    number = digit+;
    real =  ((digit)+ ’.’ (digit)*) |
            ((digit)* ’.’ (digit)+);
    whitespace = (’ ’ | ’\t’ | ’\n’)+;
    comments = (’--’ [’a’..’z’]* ’\n’);
Ignored Tokens
    whitespace,
    comments;
```

PROGRAM 2.10. SableCC specification of the tokens from Figure 2.2.

expression production defines a token, a *token-manager declaration* can be used by the generated lexical analyzer, and two other kinds are used to define the grammar from which the parser is generated.

A lexical specification uses regular-expression productions; there are four kinds: `TOKEN`, `SKIP`, `MORE`, and `SPECIAL_TOKEN`. We will only need `TOKEN` and `SKIP` for the compiler project in this book. The kind `TOKEN` is used to specify that the matched string should be transformed into a token that should be communicated to the parser. The kind `SKIP` is used to specify that the matched string should be thrown away.

In Program 2.9, the specifications of `ID`, `NUM`, and `REAL` use the abbreviation `DIGIT`. The definition of `DIGIT` is preceeded by `#` to indicate that it can be used only in the definition of other tokens.

The last part of Program 2.9 begins with `void Start`. It is a *production* which, in this case, allows the generated lexer to recognize any of the four defined tokens in any order. The next chapter will explain productions in detail.

SABLECC

The tokens described in Figure 2.2 are specified in SableCC as shown in Program 2.10. A SableCC specification file has six sections (all optional):

1. Package declaration: specifies the root package for all classes generated by SableCC.
2. Helper declarations: a list of abbreviations.

3. State declarations: support the state feature of, for example, GNU FLEX; when the lexer is in some state, only the tokens associated with that state are recognized. States can be used for many purposes, including the detection of a beginning-of-line state, with the purpose of recognizing tokens only if they appear at the beginning of a line. For the compiler described in this book, states are not needed.
4. Token declarations: each one is used to specify that the matched string should be transformed into a token that should be communicated to the parser.
5. Ignored tokens: each one is used to specify that the matched string should be thrown away.
6. Productions: are used to define the grammar from which the parser is generated.

PROGRAM

LEXICAL ANALYSIS

Write the lexical-analysis part of a JavaCC or SableCC specification for MiniJava. Appendix A describes the syntax of MiniJava. The directory

```
$MINIJAVA/chap2/javacc
```

contains a test-scaffolding file `Main.java` that calls the lexer generated by `javacc`. It also contains a `README` file that explains how to invoke `javacc`. Similar files for `sablecc` can be found in `$MINIJAVA/chap2/sablecc`.

FURTHER READING

Lex was the first lexical-analyzer generator based on regular expressions [Lesk 1975]; it is still widely used.

Computing ϵ-closure can be done more efficiently by keeping a queue or stack of states whose edges have not yet been checked for ϵ-transitions [Aho et al. 1986]. Regular expressions can be converted directly to DFAs without going through NFAs [McNaughton and Yamada 1960; Aho et al. 1986].

DFA transition tables can be very large and sparse. If represented as a simple two-dimensional matrix (*states* × *symbols*), they take far too much memory. In practice, tables are compressed; this reduces the amount of memory required, but increases the time required to look up the next state [Aho et al. 1986].

Lexical analyzers, whether automatically generated or handwritten, must manage their input efficiently. Of course, input is buffered, so that a large batch of characters is obtained at once; then the lexer can process one character at a time in the buffer. The lexer must check, for each character, whether the end of the buffer is reached. By putting a *sentinel* – a character that cannot be part of any token – at the end of the buffer, it is possible for the lexer to check for end-of-buffer only once per token, instead of once per character [Aho et al. 1986]. Gray [1988] uses a scheme that requires only one check per line, rather than one per token, but cannot cope with tokens that contain end-of-line characters. Bumbulis and Cowan [1993] check only once around each cycle in the DFA; this reduces the number of checks (from once per character) when there are long paths in the DFA.

Automatically generated lexical analyzers are often criticized for being slow. In principle, the operation of a finite automaton is very simple and should be efficient, but interpreting from transition tables adds overhead. Gray [1988] shows that DFAs translated directly into executable code (implementing states as case statements) can run as fast as hand-coded lexers. The Flex "fast lexical-analyzer generator" [Paxson 1995] is significantly faster than Lex.

EXERCISES

2.1 Write regular expressions for each of the following.

a. Strings over the alphabet $\{a, b, c\}$ where the first a precedes the first b.

b. Strings over the alphabet $\{a, b, c\}$ with an even number of a's.

c. Binary numbers that are multiples of four.

d. Binary numbers that are greater than 101001.

e. Strings over the alphabet $\{a, b, c\}$ that don't contain the contiguous substring baa.

f. The language of nonnegative integer constants in C, where numbers beginning with 0 are *octal* constants and other numbers are *decimal* constants.

g. Binary numbers n such that there exists an integer solution of $a^n + b^n = c^n$.

2.2 For each of the following, explain why you're not surprised that there is no regular expression defining it.

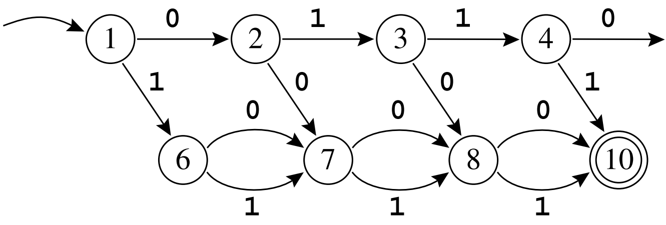
a. Strings of a's and b's where there are more a's than b's.

b. Strings of a's and b's that are palindromes (the same forward as backward).

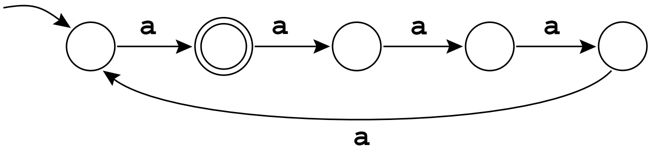
c. Syntactically correct Java programs.

2.3 Explain in informal English what each of these finite-state automata recognizes.

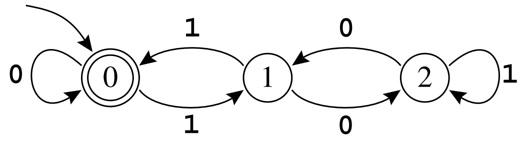
a.



b.



*c.



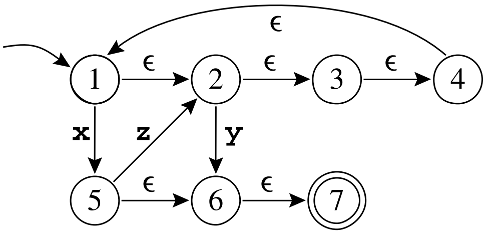
2.4 Convert these regular expressions to nondeterministic finite automata.

a. (**if**|**then**|**else**)

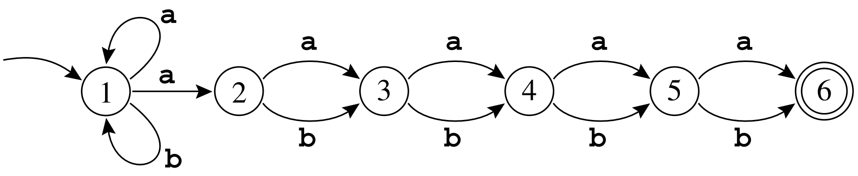
b. $\mathbf{a}((\mathbf{b}|\mathbf{a}^*\mathbf{c})\mathbf{x})^*|\mathbf{x}^*\mathbf{a}$

2.5 Convert these NFAs to deterministic finite automata.

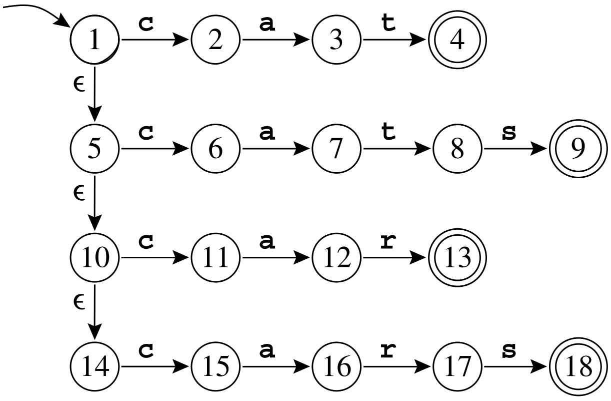
a.



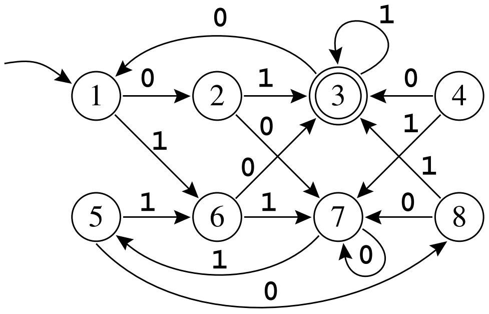
b.



c.



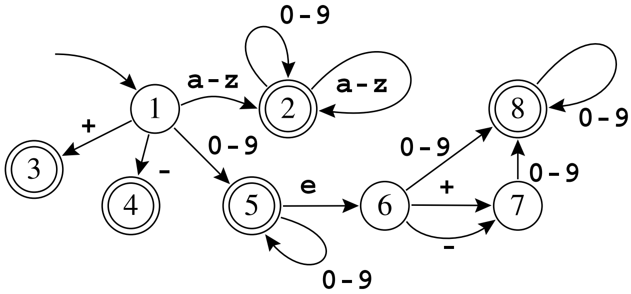
2.6 Find two equivalent states in the following automaton, and merge them to produce a smaller automaton that recognizes the same language. Repeat until there are no longer equivalent states.



Actually, the general algorithm for minimizing finite automata works in reverse. First, find all pairs of inequivalent states. States X, Y are inequivalent if X is final and Y is not or (by iteration) if $X \overset{a}{\to} X'$ and $Y \overset{a}{\to} Y'$ and X', Y' are inequivalent. After this iteration ceases to find new pairs of inequivalent states, then X, Y are equivalent if they are not inequivalent. See Hopcroft and Ullman [1979], Theorem 3.10.

***2.7** Any DFA that accepts at least one string can be converted to a regular expression. Convert the DFA of Exercise 2.3c to a regular expression. **Hint:** First, pretend state 1 is the start state. Then write a regular expression for excursions to state 2 and back, and a similar one for excursions to state 0 and back. Or look in Hopcroft and Ullman [1979], Theorem 2.4, for the algorithm.

***2.8** Suppose this DFA were used by Lex to find tokens in an input file.



a. How many characters past the end of a token might Lex have to examine before matching the token?

b. Given your answer k to part (a), show an input file containing at least two tokens such that *the first call* to Lex will examine k characters *past the end of the first token* before returning the first token. If the answer to part (a) is zero, then show an input file containing at least two tokens, and indicate the endpoint of each token.

2.9 An interpreted DFA-based lexical analyzer uses two tables,

`edges` indexed by state and input symbol, yielding a state number, and
`final` indexed by state, returning 0 or an action-number.

Starting with this lexical specification,

```
(aba)+      (action 1);
(a(b*)a)    (action 2);
(a|b)       (action 3);
```

generate the `edges` and `final` tables for a lexical analyzer.

Then show each step of the lexer on the string `abaabbaba`. Be sure to show the values of the important internal variables of the recognizer. There will be repeated calls to the lexer to get successive tokens.

****2.10** Lex has a *lookahead* operator / so that the regular expression `abc/def` matches `abc` only when followed by `def` (but `def` is not part of the matched string, and will be part of the next token(s)). Aho et al. [1986] describe, and Lex [Lesk 1975] uses, an incorrect algorithm for implementing lookahead (it fails on `(a|ab)/ba` with input `aba`, matching `ab` where it should match `a`). Flex [Paxson 1995] uses a better mechanism that works correctly for `(a|ab)/ba` but fails (with a warning message) on `zx*/xy*`.

Design a better lookahead mechanism.